

# RP-Filter: A Path-Based Triple Filtering Method for Efficient SPARQL Query Processing

Kisung Kim<sup>1</sup>, Bongki Moon<sup>2</sup>, and Hyoung-Joo Kim<sup>1</sup>

<sup>1</sup> Seoul National University, Seoul, Korea  
kskim@idb.snu.ac.kr, hjk@snu.ac.kr

<sup>2</sup> University of Arizona, Tucson, U.S.A  
bkmoon@cs.arizona.edu

**Abstract.** With the rapid increase of RDF data, the SPARQL query processing has received much attention. Currently, most RDF databases store RDF data in a relational table called triple table and carry out several join operations on the triple tables for SPARQL query processing. However, the execution plans with many joins might be inefficient due to a large amount of intermediate data being passed between join operations. In this paper, we propose a triple filtering method called RP-Filter to reduce the amount of intermediate data. RP-Filter exploits the path information in the query graphs and filters the triples which would not be included in final results in advance of joins. We also suggest an efficient relational operator RFLT which filters triples by means of RP-Filter. Experimental results on synthetic and real-life RDF data show that RP-Filter can reduce the intermediate results effectively and accelerate the SPARQL query processing.

**Keywords:** RDF store, SPARQL query processing, triple filtering, intermediate results.

## 1 Introduction

RDF(Resource Description Framework)[1] is the standard data model recommended by W3C for the sake of describing data in the semantic web. RDF data is a set of triples(subject, predicate, object) which describe the relationship between two resources(subject and object). The RDF data forms a graph called RDF graph which consists of the resources and their relationships. SPARQL[2] is the standard query language for RDF data and expresses the user's data needs as graph patterns. The SPARQL query processing can be viewed as the sub-graph pattern matching problem for the RDF graph[3]. RDF features flexibility with little schema restriction and expressive power which can represent graph-structured data. By virtue of these features, RDF is widely used in many areas. For example, RDF has been used for the purpose of integrating heterogeneous databases or publishing data on the web in many areas, e.g. life science[4,5], open government[6], social networking[7] and multimedia[8].

With the fast growth of RDF data, there has been a lot of research on storing and querying of RDF data[9,10,11,12]. Most state-of-the-art RDF engines employ

the relational model to store and manipulate RDF data. They store RDF data in a relation with three columns(S,P,O) which is called a *triple table* and evaluate SPARQL queries through a sequence of joins on the triple table. Let us consider the following SPARQL query.

```
SELECT ?n1 ?n2 ?n3 ?n4
WHERE {?n1 <p1> ?n2.
       ?n2 <p2> ?n3.
       ?n3 <p3> ?n4.}
```

The above SPARQL query consists of three triple patterns, which form a graph pattern. The evaluation of the SPARQL query is to find all subgraphs in the RDF graph matching with the query graph pattern. Fig. 1(a) shows a possible execution plan for the SPARQL query, which have three scan operators(one for each triple pattern) and two join operators. Each operator in the execution plans makes the partially matching fragments for the query graph pattern. For example, *Join<sub>1</sub>* in Fig. 1(a) produces all the matching fragments for the graph pattern which consists of the second and the third triple pattern of the SPARQL query.

This form of the execution plan is widely adopted by many RDF engines but has a problem that it might generate many useless intermediate results. The useless intermediate results are the results which are generated by some operators but not included in the final results of the query. Assume that the numbers in Table 1 are the result cardinalities for the subgraph patterns included in the query graph pattern. *Join<sub>1</sub>* in Fig. 1 (a) generates all the matching fragments for the graph pattern in the third row of Table. 1 and the number of the result rows is 500,000. However, the number of the final results(the first row in Table. 1) is only 1,000. Consequently, at least 499,000 rows of 500,000 rows become the useless intermediate results. The cost which are consumed for generating and processing them is wasted because the useless intermediate results do not contribute to the query results. And in large-scale RDF dataset, the size of the intermediate results intends to increase and the overhead for the useless intermediate results becomes more serious.

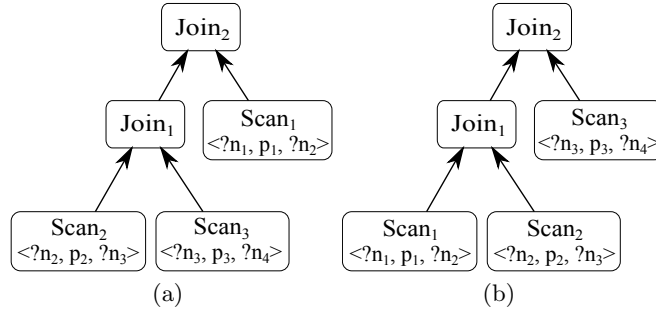
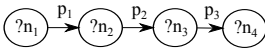
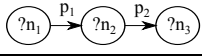
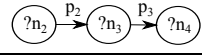


Fig. 1. Execution Plan

**Table 1.** Cardinalities of Intermediate Results

Graph Pattern	Cardinality
	1,000
	1,000,000
	500,000

Most RDF engines try to reduce these intermediate results by choosing an execution plan with the optimal join order when compiling the query. For example, Fig. 1(b) shows another execution plan whose results are the same with the Fig. 1(a) but whose join order is different from that of the execution plan in Fig. 1(a). The query optimizer prefers the execution plan in Fig. 1(a) to the execution plan in Fig. 1(b) because the latter would generate 500,000 more rows than the former plan. However, as we can see in this example, the execution plan with the optimal join order could not remove all the useless intermediate results.

In this paper, we propose a novel triple filtering method called *RP-Filter* (RDF Path Filter) to reduce the useless intermediate results effectively and efficiently using graph-structural information of RDF data. RP-Filter provides the list of the nodes in the RDF graph which are reached by paths with a specific path pattern. For example, we can obtain the list of nodes which can be reached by paths which are matching for a path pattern  $\{(?n_1, p_1, ?n_2), (?n_2, p_2, ?n_3)\}$  from RP-Filter. This node list can be used as filter data to filter the result triples of *Scan<sub>2</sub>* or *Scan<sub>3</sub>* in Fig. 1(a) and then we can prune the triples which would not be joined in *Join<sub>2</sub>* in advance. As a result, we can reduce the number of intermediate results using the path pattern information. Through these node lists, we can reduce the useless intermediate results effectively for complex SPARQL queries.

RP-Filter utilizes some properties of RDF engines, one of which is that they store the triples in a sorted order and the scanned triples are also sorted. Many RDF engines store their triples as sorted because it give many optimization opportunities. The filtering process of RP-Filter is very efficient and incurs little overhead to the normal query processing because it utilizes this ordering property of scanned triples.

We propose the definition of RP-Filter and an relational operator called RPFLT which filters the input triples using RP-Filter. We also propose a method to generate an execution plan with the RPFLT operators using heuristic method. We carried out with several queries on large real-life and benchmark RDF datasets to evaluate RP-Filter. These results demonstrate that RP-Filter effectively and efficiently reduces the useless intermediate results and consequently accelerates the SPARQL query evaluation.

## 2 Related Work

Early RDF systems provide storage systems which use row-oriented RDBMSs as their back-end storages :e.g like Jena[10] and Sesame[9] (currently, Jena and Sesame provide also native storage systems which do not use RDBMSs). These RDBMS-based RDF systems store RDF triples in a triple table and utilize the query processing modules of RDBMSs when processing RDF queries. However, RDBMSs are not optimized to store and query the graph-structured RDF data and have several scalability problems.

SW-Store[11] uses a column-oriented store as its underlying store and propose the vertical partitioning method. SW-Store partitions the triple table by the predicate column and shows that the partitioning of the triple table have many advantages, like reduced I/O costs and the compact storage size. In addition, the triples are stored as sorted in a column-oriented store so that fast merge joins can be used when processing a query.

Currently the fastest RDF engine according to the published numbers is RDF-3X[12]. It stores triples in six clustered indexes redundantly as sorted by six different orderings(SPO,SOP,PSO,POS,OSP,OPS). RDF-3X can read matching triples for any type of triple patterns sorted by any ordering using the six indexes. Also RDF-3X uses a block compression techniques which store only deltas between triples rather than writing actual values. The compression technique reduces the size of storage and the number of disk I/O requests needed for reading triples. RDF-3X generates an execution plan which consists of mainly scan operators and join operators for a SPARQL query. Each scan operator in an execution plan scans one of the six indexes and the ordering of scanned triples depends on which index is used. There are two types of join operators: merge join and hash join. RDF-3X uses a merge join when the orderings of two input relations are the same with join variable. Otherwise RDF-3X uses a hash join.

In order to reduce the useless intermediate results, the authors of RDF-3X propose an RDF-specific *sideway information passing* technique called U-SIP[13]. It builds a sort of filters while processing an operator of an execution plan and passes the filters to other operators to avoid generating the useless intermediate results. U-SIP exploits the pipelined data flow of an execution plan to pass the filter information. However, it cannot transfer the filter information reversely to the data flow of the execution plan. As a result, the cases where U-SIP can be applied are limited by the pipeline-blocking operator like a hash join. Especially, the execution plan for long path patterns would use many hash joins and U-SIP could not be very effective in these cases.

## 3 Preliminary

In this section, we describe RDF data model and SPARQL query model. We do not cover the entire RDF and SPARQL specification. Rather we deal with a core fragment of RDF and SPARQL. We do not consider blank nodes, literals and data types in RDF. For SPARQL, we concentrate on the graph pattern

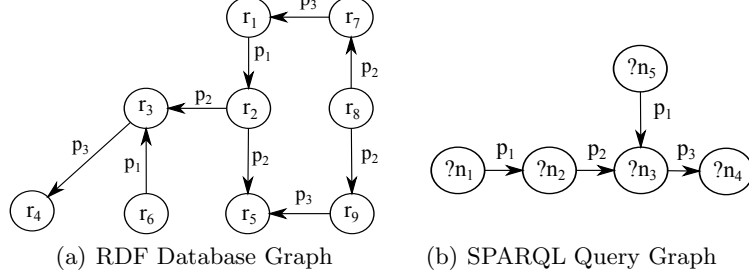


Fig. 2. RDF Database Graph and SPARQL Query Graph

matching of SPARQL, more specifically basic graph patterns[2] which consists of only conjunctive triple patterns. We also do not consider a join with predicate or a triple pattern having a variable predicate, as they are rarely used.

We assume the existence of two pairwise disjoint sets: a set  $U$  of URIs and a set  $VAR$  of variables. A variable symbol starts with  $?$  to distinguish with URIs. A triple,  $t \in U \times U \times U$  is called an RDF triple, and a triple,  $tp \in (U \cup VAR) \times (U \cup VAR) \times (U \cup VAR)$  is called a triple pattern.

An RDF database  $D$  is a finite set of triples, and a SPARQL query  $Q$  is a finite set of triple patterns. We define three subsets of  $U$  such that  $S = \{s | s \in U \wedge \exists t(s, p, o) \in D\}$ ,  $P = \{p | p \in U \wedge \exists t(s, p, o) \in D\}$ ,  $O = \{o | o \in U \wedge \exists t(s, p, o) \in D\}$ .

We map RDF database  $D$  into a graph  $G_D = (N_D, E_D, L_D)$  which consists of a node set  $N_D$ , a edge set  $E_D$  and a label set  $L_D$ , where  $N_D = S \cup O$ ,  $E_D = \{(s, p, o) | t(s, p, o) \in D\}$  and  $L_D = P$ . A SPARQL query  $Q$  is also mapped into a graph  $G_Q = (N_Q, E_Q, L_Q)$ , where  $N_Q \subseteq S \cup O \cup VAR$ ,  $E_Q = \{(s, p, o) | t(s, p, o) \in Q\}$  and  $L_Q \subseteq P$ . Both  $G_D$  and  $G_Q$  are edge labeled directed graphs. Fig. 2(a) shows an example RDF database graph and Fig. 2(b) shows an example SPARQL query graph.

A *path* on a graph  $G = (N, E, L)$  is a sequence of connected edges in the graph. If the terminal node of a path is  $n$ , the path is an *incoming path* of  $n$ . For example, for the SPARQL query graph in Fig. 2(b),  $pa = \langle (?n_1, p_1, ?n_2), (?n_2, p_2, ?n_3) \rangle$  is an incoming path of  $?n_3$ .

We define a *predicate path* as a sequence of predicates. The predicate path of a given path in  $G = (N, E, L)$  is a sequence of edge labels of the path. For example, the predicate path of  $pa$  is  $\langle p_1, p_2 \rangle$ .

We use the notations  $PPath(p)$  and  $|PPath(p)|$  to denote the predicate path of a path  $p$  and its length, respectively. We also use  $InPPath(n)$  to denote a set of all incoming predicate paths of  $n \in N$ . When the maximal path length  $l$  is given, a variant of the notation,  $InPPath(n, l)$ , is used to denote a subset of  $InPPath(n)$  such that  $InPPath(n, l) = \{ppath | ppath \in InPPath(n) \wedge |ppath| \leq l\}$ .

*Example 1 (Incoming Predicate Path).* For the SPARQL query graph in Fig. 2(b), the incoming path set of  $?n_4$  with the maximum length 2 is  $InPPath(?n_4, 2) = \{\langle p_3 \rangle, \langle p_1, p_3 \rangle, \langle p_2, p_3 \rangle\}$ .

## 4 RP-Filter

In this section, we present the definition of RP-Filter and the physical storage model of RP-Filter. To begin with, we discuss the requirement of RDF stores to use RP-filter.

### 4.1 Requirements of RP-Filter

In order to apply the RP-Filter technique into an RDF engine, the RDF engine should meet the following requirements.

1. URIs are mapped into integer IDs and the triples are stored using the IDs in a triple table with three column, S,P and O.
2. The execution plan has one scan operator for each triple pattern in the SPARQL query, which reads triples matching with the triple pattern from disks.
3. The scan operators read triples as sorted by the S or the O column.

Several RDF engines including RDF-3X utilize storage and query processing techniques which meet three conditions above for efficient query processing. Especially, the third condition is relatively strict. However, the condition is satisfied by several RDF store, e.g. RDF-3X and SW-store. This is because that the sorted materialization of triples provides a lot of efficiency, like the fast retrieval of matching triples and the usage of fast merge join. Therefore, RP-Filters can be adopted by various RDF engines including RDF-3X and SW-store.

### 4.2 Definition of RP-Filter

The RP-Filter for an RDF database is a set of node lists. A node list for a predicate path contains all the node IDs which have the predicate path as its incoming predicate path.

**Definition 1 (Node List of  $ppath$  N-List( $ppath$ )).** *The node list for a predicate path  $ppath$  is a sorted list of IDs of nodes  $n$  which satisfy that  $ppath \in InPPath(n)$ . We denote the node list of a predicate path  $ppath$  as  $N-List(ppath)$ .*

By reading the node list for a predicate path, we can easily get all the node IDs which are reached by the path pattern which is described by the predicate path. Note that the node lists are sorted by the node IDs. The RP-Filter for an RDF database is defined as follows.

**Definition 2 (RP-Filter of RDF database  $D$  with the maximum length  $MaxL$ ).** *Given an RDF database  $D$  and the maximum length  $MaxL$ , the RP-Filter of  $D$  is a set of all pairs  $\langle ppath, N-List(ppath) \rangle$ , for all  $ppaths$  which exist in  $D$  and whose lengths are less than or equal to  $MaxL$ .  $RP-Filter(D, MaxL) = \{ \langle ppath, N-List(ppath) \rangle \mid ppath \in \bigcup_{i=1}^{maxL} P^i \wedge ppath \text{ exists in } D \}$ .*

ppath	N-List	ppath	N-List
$\langle p_1 \rangle$	$\boxed{r_2} \boxed{r_3}$	$\langle p_2, p_3 \rangle$	$\boxed{r_1} \boxed{r_4} \boxed{r_5}$
$\langle p_2 \rangle$	$\boxed{r_3} \boxed{r_5} \boxed{r_7} \boxed{r_9}$	$\langle p_3, p_1 \rangle$	$\boxed{r_2}$
$\langle p_3 \rangle$	$\boxed{r_1} \boxed{r_4} \boxed{r_5}$	$\langle p_1, p_2, p_3 \rangle$	$\boxed{r_4}$
$\langle p_1, p_2 \rangle$	$\boxed{r_3} \boxed{r_5}$	$\langle p_2, p_3, p_1 \rangle$	$\boxed{r_2}$
$\langle p_1, p_3 \rangle$	$\boxed{r_4}$	$\langle p_3, p_1, p_2 \rangle$	$\boxed{r_3} \boxed{r_5}$

**Fig. 3.** RP-Filter (MaxL=3)

We say that a predicate path  $ppath$  exists in  $D$  if and only if there exists a path whose predicate path is  $ppath$  in  $D$ . We introduce  $MaxL$  to limit the size of RP-Filters. As  $MaxL$  increases, the number of predicate paths increases. As a result, the applicable RP-Filters also increase and the quality of RP-Filter improves but the size of RP-Filter also increases. In other words, there exists a tradeoff between the quality of RP-Filter and the space overhead of RP-Filter. We can control the tradeoff using the MaxL value.

*Example 2 (RPFILTER).* Fig. 3 show RP-Filter( $D, 3$ ) for the RDF database  $D$  in Fig. 2(a). The figure shows predicate paths and their node lists. There are 11 node lists in RP-Filter( $D, 3$ ). And we can see that each node list is sorted by the node IDs.

### 4.3 Storage Model of RP-Filter

Each node list is stored in disk as sorted by node IDs so that the node list can be read in the sorted order while processing triple filtering. We use the delta based block compression technique used in RDF-3X [12] to alleviate the size overhead of RP-Filter and the disk I/O overhead for reading the node lists. In this method, the difference between two adjacent node IDs are stored. According to the size of the delta, the size of bytes for the writing of the delta is determined. We can use this compression method because the node IDs are sorted.

We organize the predicate paths in a trie(or prefix tree) called RP-Trie in order to search the node lists for a predicate path efficiently. RP-Trie is a trie built with all the predicate paths in RP-Filter. Each node in level  $l$  in RP-Trie has a pointer to the node list for its associated length- $l$  predicate path. Fig. 4 shows RP-Trie for RP-Filter( $D, 3$ ) in Fig. 3. We can find the location in disk of the node list for a predicate path by traversing RP-Trie using the predicate path. If there is no node for a predicate path whose length  $\leq MaxL$ , we can conclude that there exists no path which is matched to the predicate path in RDF database.

RP-Trie has the worst case space complexity,  $O(\sum_{i=1}^{MaxL} |P|^i)$  and can grow exponentially to  $MaxL$ . But as we can see in section 6, for the real-life data sets and small  $MaxL$  value, the size of RP-Trie is relatively of small size and the RP-Trie can be resident in the main memory.

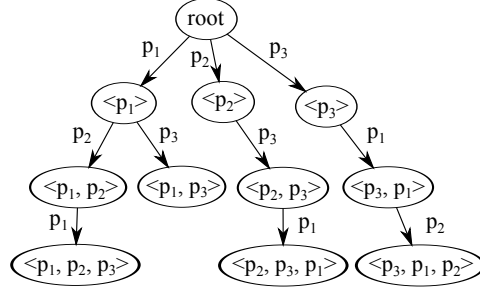


Fig. 4. RP-Trie for RP-Filter( $D, 3$ )

The structure of RP-Filter resembles the inverted index structure which is widely used in the information retrieval area. Each predicate path in RP-Filter can be considered a lexicon of an inverted index, while the node list is pretty much like a posting list. Just as a posting list of an inverted index keeps document IDs in sorted order, the node list keeps the node IDs in the sorted order. Note that the predicate paths are organized into RP-Trie to assist in locating the node list for a predicate path and finding relevant node IDs quickly.

## 5 Query Evaluation Using RP-Filter

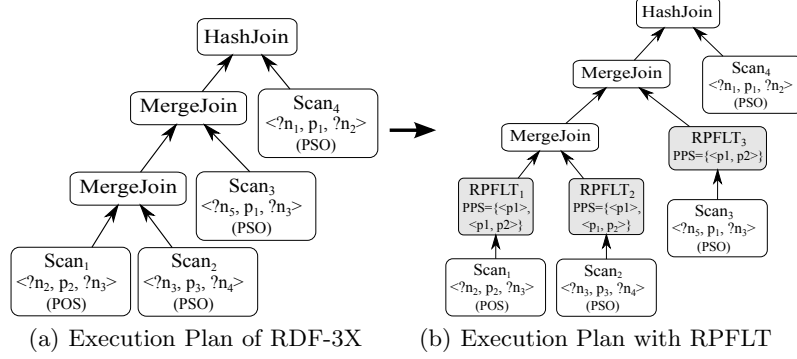
In this section, we introduce a filtering operator RPFLT and then we discuss the query plan generation with the RPFLT operator.

### 5.1 RPFLT Operator

RP-Filter is used to filter the triples from scan operators in an execution plan. In order to use RP-Filter, the query compiler adds an operator called RPFLT to an execution plan. The RPFLT operator is a relational operator which gets triples from its child scan operator and outputs only the triples passing RP-Filter. An RPFLT operator is added to an execution plan as a parent operator of a scan operator.

**Predicate Path Set of RPFLT.** An RPFLT operator has a set of predicate paths called *PPS* (Predicate Path Set) assigned by the query compiler. To explain which predicate paths can be included in the PPS of an RPFLT, we define a property of a scan operator called *sortkey* as follows. The result triples of a scan operator are ordered by the S or the O column (not by the predicate column because we do not consider the predicate variable and the predicate join). We call the column by which the result triples are sorted a *sortkey column* of the scan operator. The sortkey column has a corresponding node in the mapped query graph. We also use the term *sortkey node* to indicate the sortkey column's





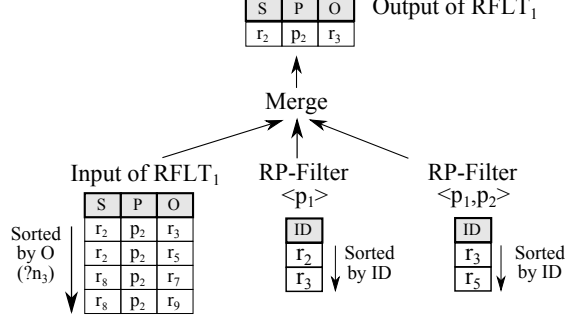
**Fig. 5.** Application of RPFLT Operators

corresponding node in the query graph.  $Scan_i.sortkey$  is used to denote the sortkey column or the sortkey node of  $Scan_i$  depending on context.

Fig. 5(a) shows an example execution plan of RDF-3X for the query graph in Fig. 2(b). The last item in each scan operator is the type of index to be used. For example,  $Scan_1$  scans the POS index and so the scanned triples are ordered by (P,O,S). Since all the triples have the same predicate values ‘ $p_2$ ’, they are actually sorted by the O column. In the same way, the results of  $Scan_2$  are ordered by the S column. Therefore, the sortkey column of  $Scan_1$  is ‘O’ and the sortkey column of  $Scan_2$  is ‘S’. And the sortkey nodes of  $Scan_1$  and  $Scan_2$  are both node  $?n_3$  in Fig. 2(a), i.e.,  $Scan_1.sortkey = Scan_2.sortkey = ?n_3$ .

The PPS of the RPFLT for  $Scan_i$  should be a subset of  $InPPath(Scan_i.sortkey, MaxL)$ . For example,  $\langle p_1, p_2 \rangle$  can be included in the PPS of the RPFLT for  $Scan_1$  because  $Scan_1.sortkey = ?n_3$  and  $\langle p_1, p_2 \rangle$  is in  $InPPath(?n_3, 3)$ . Fig. 5(b) shows an execution plan which uses two RPFLT operators. The last item of an RPFLT operator lists the predicate paths in its PPS. The RPFLT operators for  $Scan_1$  and  $Scan_2$  have the same PPS because the sortkey for the two scan operators are same. The PPS is  $\{\langle p_1, p_2 \rangle, \langle p_1 \rangle\}$  but  $InPPath(?n_3, 3)$  is  $\{\langle p_1, p_2 \rangle, \langle p_1 \rangle, \langle p_2 \rangle\}$ . The reason why only two of three predicate paths are included in the PPS is described in section 5.2.  $Scan_3$  operator does not have an RPFLT operator because its sortkey node has no incoming predicate path.

**Filtering Process of RPFLT.** An RPFLT operator outputs the triples whose values of the sortkey column are include in all N-Lists for the predicate paths in its PPS. Fig. 6 illustrates the filtering process of  $Scan_1$  in Fig. 5(b). The filtering process merges the input triples with all assigned N-Lists. In this example,  $Scan_1$  outputs four triples but three of them are filtered out by  $RPFLT_1$ .  $RPFLT_1$  outputs only one triple whose object is ‘ $r_3$ ’ because ‘ $r_3$ ’ is in both N-List( $\langle p_1, p_2 \rangle$ ) and N-List( $\langle p_1 \rangle$ ).

**Fig. 6.** Filtering in RPFLT Operator

The  $?n_3$  node( $Scan_1.sortkey$ ) in the query graph has the two predicate paths in its InPPath. So the matching data nodes for  $?n_3$  must have the two incoming predicate paths, too. The intersection of the two N-Lists gives us the matching data nodes which have both of the two incoming predicate paths. If the three filtered triples were not filtered out, they would be carried over to the next join operations - a *MergeJoin* and a *HashJoin* - and slow down the overall query processing without contributing to the final query result. In this manner, we attempt to filter triples out at the earliest possible stage, if they would not be included in the final results. Note that the filtering process involves reading the node lists and merging them with the input triples. We can filter the input triples simply by merging the node lists and the input triples because they share the same orderings. Also the N-Lists are usually of small length. Consequently, the reading and merging of the N-Lists incur little overhead and the RPFLT operator is very efficient and light operator.

**Analysis of RPFLT.** If we consider an node list as a table with the single column ID, the output of RPFLT can be described formally as following.

$$RPFLT(Scan_i, PPS) = \left( \bigcap_{ppath \in PPS} \text{N-List}(ppath) \right) \bowtie_{ID=Scan_i.sortkey} Scan_i \quad (1)$$

We use  $Scan_i$  to denote the result relation of  $Scan_i$ , which have three columns and the  $Scan_i.sortkey$  to denote the sortkey column of  $Scan_i$ . Note that we use the relational algebra only to describe the output of RPFLT not to describe the evaluation order of RPFLT.

The cost of  $RPFLT(Scan_i, PPS)$  is  $B \times \sum_{ppath \in PPS} \|\text{N-List}(ppath)\| + C \times \left( \sum_{ppath \in PPS} |\text{N-List}(ppath)| + |Scan_i| \right)$ , where  $B$  is the cost of disk block I/O,  $C$  is the cpu cost for the merging,  $|\text{N-List}(ppath)|$  and  $\|\text{N-List}(ppath)\|$  are the number of nodes in  $\text{N-List}(ppath)$  and the number of blocks of  $\text{N-List}(ppath)$ , respectively.

## 5.2 Generating an Execution Plan with RPFLT Operators

We use the 2-phase query optimization method to make execution plans with RPFLT operators. In the first phase, the query compiler generates an optimized execution plan through its normal query optimization process. Then, in the second phase, the query compiler adds RPFLT operators to the optimized execution plans. This 2-phase method uses heuristics that the optimized plan with no RPFLT also tends to be optimal when augmented by the RPFLT operators.

In the second phase, the query compiler examine the incoming predicate paths of the sortkey node of each scan operator in the execution plan. The query compiler makes decisions about which predicate paths are included in the PPS for the RPFLT operator. If the PPS is empty, no RPFLT operator is added to the scan operator.

When deciding the PPS, the query compiler should be careful not to choose redundant predicate paths. If a predicate path  $ppath_1$  is a suffix of another predicate path  $ppath_2$ ,  $N\text{-List}(ppath_1) \supset N\text{-List}(ppath_2)$ . Therefore, it is of no use to include both  $ppath_1$  and  $ppath_2$  in PPS. The query compiler should include only  $ppath_2$  in PPS because  $N\text{-List}(ppath_2)$  has less node IDs than  $N\text{-List}(ppath_1)$  and is more effective filter. For example, let us take a look at Fig. 5 again. There exist three predicate paths in  $\text{InPPath}(?n_3, 3) = \{\langle p_1, p_2 \rangle, \langle p_1 \rangle, \langle p_2 \rangle\}$ . However, we do not include  $\langle p_2 \rangle$  in the PPS of  $Scan_1$  (or  $Scan_2$ ), because  $\langle p_2 \rangle$  is a suffix of  $\langle p_1, p_2 \rangle$ .

For another case of redundant predicate paths, if the triple pattern of  $Scan_i$  is  $\langle ?s, p_n, ?o \rangle$ , we need not to include  $\langle p_n \rangle$  in the PPS even though  $\langle p_n \rangle$  is in  $\text{InPPath}$ . The reason is because  $N\text{-List}(\langle p_n \rangle)$  could not filter any triple from  $Scan_i$ .

Note that the execution plan generated by this 2-phase method might not optimal. It is because RPFLT operators can change the cardinalities of the intermediate results. The join order of the original execution plan might be not optimal for the changed cardinalities of the intermediate results. To solve this problem, we should be able to decide the optimal join order for the execution plan with RPFLT. However, this requires a method to estimate the cardinalities of the filtered triples. Also, the additional costs of RPFLT operators could make the execution plan more expensive than the original execution plan. However, we leave this issue as future work and here we use this heuristic method.

## 6 Experimental Results

We implemented RP-Filter on the open-source system RDF-3X(0.3.5 version). The system was implemented with C++ and compiled by g++ with -O3 flag. We conducted all the experiments on an IBM machine having 8 3.0GHz Intel Xeon cores, 16GB memory and running 64bit 2.6.31-23 Linux Kernel. We used two datasets: YAGO2[14] as a small real-life dataset and LUBM[15] as a large synthetic dataset. We generated the LUBM dataset with 10,000 universities<sup>1</sup>.

<sup>1</sup> <http://swat.cse.lehigh.edu/projects/lubm/>

**Table 2.** Statistics about Datasets

	predicate	triples	RDF-3X size
YAGO2	94	195,048,649	7.6 GB
LUBM	18	1,334,681,192	70 GB

Table 2 shows statistics about the datasets. Note that YAGO2 has 94 predicates but LUBM has only 18 predicates. That is because YAGO2 is a combination of heterogeneous datasets(Wordnet and Wikipedia), while LUBM is about relatively homogeneous domain(university).

### 6.1 RP-Filter Size

For two datasets we built RP-Filters with MaxL=3. Table 3 shows the number of N-Lists and the total size of RP-Filters. As we can see, the size of RP-Filters for two datasets are much smaller than the input dataset sizes. The number of N-Lists for YAGO2 is higher than that of LUBM, although the data size of YAGO2 is smaller than the size of LUBM. This is because YAGO2 has more predicates than LUBM, there exists more distinct predicate paths in YAGO2.

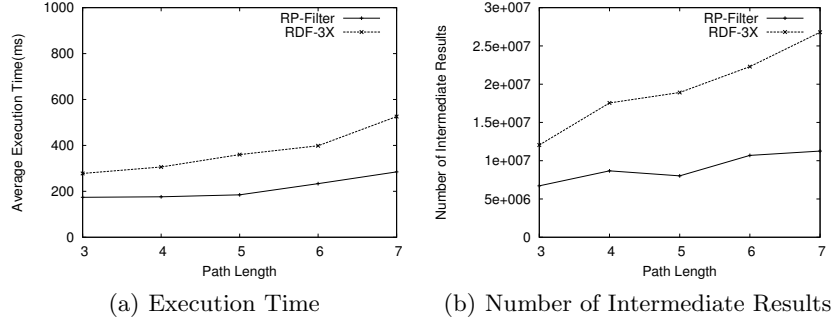
### 6.2 Query Execution Time

To evaluate the performance of RP-Filter, we compared the query execution time of RDF-3X using RP-Filter with the original RDF-3X system. We measured the executions in the wall-clock time. RDF-3X converts node IDs in the final results into URIs to display the query results in the final stage of the query evaluation. The converting process is very time-consuming when there are large number of final results. Because the converting process is not relevant to the performance evaluation of RP-Filter, we excluded it from the execution time. We also counted the number of intermediate results for each evaluation. The number of intermediate results is the summation of the number of results of all the operators in the execution plan except the final operator. We exclude it because the number of final results is not changed by the filtering.

**YAGO2 Dataset.** For YAGO2 dataset, we generated several random path queries. We chose 15 predicates and made 3~7-length path queries using the chosen predicates. Each path queries has a single path and similar to the SPARQL query in Section 1.

**Table 3.** RP-Filter Size(MaxL=3)

	N-Lists	total size	avg. length
YAGO2	39,635	836MB	16,305
LUBM	122	1.3GB	2,571,504



**Fig. 7.** Evaluation Results:YAGO

Fig. 7 shows the average execution time and the average number of the intermediate results. The average execution time of RP-Filter is lower than that of RDF-3X. Also the number of the total intermediate results is lower than that of RDF-3X. The growth rate of the execution time and the number of the intermediate results for RP-Filter is slower than those of RDF-3X.

**LUBM Dataset.** For LUBM datasets, we used two queries from 14 queries of the LUBM benchmark. We chose two of them because other queries have very simple structure and short paths in them. The queries we chose are Q2 and Q9. They have a 3-length path and complex structures. Q2 and Q9 are listed in Appendix. In order to execute the queries, we had to change the query Q9 slightly. In fact, LUBM is a benchmark for the RDF engines with the reasoning capability. However, currently RDF-3X does not support RDF reasoning. RDF-3X gives no answer for the query Q9 because the query asks about the instances of an inferred class. Therefore, we changed the class name so that no inference is needed, while leaving the structure of the queries unchanged.

Table 4 shows the execution time and the number of intermediate results for each query. For cold cache, the file system caches were dropped by `/bin/sync` and `echo 3 > /proc/sys/vm/drop_caches` commands. We evaluated the queries with U-SIP technique in RDF-3X. The results show that U-SIP is not very effective for the queries we used. The reason is that the execution plans involve many hash joins so the effect of U-SIP is limited. For RP-Filter, Q2 and Q9 showed different results. Q2 was improved by a factor of about 3 but for Q9 RP-Filter had little effect. That is because in Q2 there exists very selective path pattern but there is no such path pattern in Q9. And For Q2 the intermediate results are significantly reduced but for Q9, the intermediate results are not reduced much. For Q9, we can also observe that the query time is slightly longer when using RP-Filter. That is because the overhead for RP-Filter is more strong than the benefits of reduced intermediate results.

**Table 4.** Evaluation Results:LUMB (times in second)

	Warm cache		Cold cache		Intermediate Results	
	q2	q9	q2	q9	q2	q9
RDF-3X	26.5	37.0	28.7	43.4	424,747,108	659,968,158
NO U-SIP	22.9	31.9	25.0	38.4	424,785,330	662,615,314
NO U-SIP RP-Filter	7.4	32.7	9.6	40.2	308,143,082	620,276,418
RP-Filter	8.1	36.9	9.3	43.7	233,654,645	617,592,582

## 7 Conclusions and Future Work

In this paper, we propose a triple filtering method called RP-Filter. Based on the information about incoming paths of the query graph, RP-Filter prunes the scanned triples which would not be included in the final results. This triple filtering helps to reduce the useless intermediate results and reducing the intermediate results improves the query execution performance. Our experimental results shows that RP-Filter is very effective to reduce the useless intermediate results. For future work, we plan to explore how to reduce the overhead of RP-Filters and how to generate plans using RP-Filters based on the cost model.

**Acknowledgements.** This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST) (No. 20110017480) and the Brain Korea 21 Project.

## Appendix

**LUBM Queries** PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
PREFIX lubm:<http://http://www.lehigh.edu#>

### Q2

```
SELECT ?x ?y ?z WHERE {
  ?x rdf:type lubm:GraduateStudent.
  ?y rdf:type lubm:University.
  ?z rdf:type lubm:Department.
  ?x lubm:memberOf ?z.
  ?z lubm:subOrganizationOf ?y.
  ?x lubm:undergraduateDegreeFrom ?y.}
```

### Q9

```
SELECT ?x ?y ?z WHERE {
  ?x rdf:type lubm:Student.
  ?y rdf:type lubm:Faculty.
  ?z rdf:type lubm:Course.
  ?x lubm:advisor ?y.
  ?x lubm:takesCourse ?z.
  ?y lubm:teacherOf ?z.}
```

## References

1. Klyne, G., Carroll, J.J.: Resource description framework (rdf): Concepts and abstract syntax. Technical report, W3C Recommendation (2004)
2. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. Technical report, W3C Recommendation (2008)
3. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. *ACM Trans. Database Syst.* 34(3) (2009)
4. Belleau, F., Nolin, M.-A., Tourigny, N., Rigault, P., Morissette, J.: Bio2RDF: towards a mashup to build bioinformatics knowledge systems. *Journal of Biomedical Informatics* 41(5), 706–716 (2008)
5. Redaschi, N., Consortium, U.: UniProt in RDF: Tackling Data Integration and Distributed Annotation with the Semantic Web. In: *Nature Precedings* (2009)
6. Sheridan, J.: Linking UK government data. In: *WWW Workshop on Linked Data*, pp. 1–4 (2010)
7. Mika, P.: Social Networks and the Semantic Web. In: *Proceedings of International Conference on Web Intelligence (WI 2004)*, pp. 285–291 (2004)
8. Kobilarov, G., Scott, T., Raimond, Y., Oliver, S., Sizemore, C., Smethurst, M., Bizer, C., Lee, R.: Media Meets Semantic Web — How the BBC uses dbpedia and Linked Data to make Connections. In: Aroyo, L., Traverso, P., Ciravegna, F., Cimiano, P., Heath, T., Hyvönen, E., Mizoguchi, R., Oren, E., Sabou, M., Simperl, E. (eds.) *ESWC 2009. LNCS*, vol. 5554, pp. 723–737. Springer, Heidelberg (2009)
9. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: Horrocks, I., Hendler, J. (eds.) *ISWC 2002. LNCS*, vol. 2342, pp. 54–68. Springer, Heidelberg (2002)
10. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: *Proceedings of the 13th International World Wide Web Conference (WWW 2004)*, pp. 74–83 (2004)
11. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal* 18(2), 385–406 (2009)
12. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for rdf. *PVLDB* 1(1), 647–659 (2008)
13. Neumann, T., Weikum, G.: Scalable join processing on very large rdf graphs. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2009)*, pp. 627–640 (2009)
14. Hoffart, J., Suchanek, F.M., Berberich, K., Lewis-Kelham, E., de Melo, G., Weikum, G.: Yago2: Exploring and querying world knowledge in time, space, context, and many languages. In: *Proceedings of the 20th International Conference on World Wide Web (WWW 2011)*, pp. 229–232 (2011)
15. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3(2-3), 158–182 (2005)